Pre-print version

R. Jhawar, V. Piuri, "Fault Tolerance and Resilience in Cloud Computing Environments", in Computer and Information Security Handbook, 2nd Edition, J. Vacca (ed.), Morgan Kaufmann, 2013. ISBN: 978-0-1239-4397-2 (to appear)

Fault Tolerance and Resilience in Cloud Computing Environments

Ravi Jhawar and Vincenzo Piuri

Abstract

The increasing demand for flexibility and scalability in dynamically obtaining and releasing computing resources in a cost-effective and device-independent manner, and easiness in hosting applications without the burden of installation and maintenance has resulted in a wide adoption of the Cloud computing paradigm. While the benefits are immense, this computing paradigm is still vulnerable to a large number of system failures and, as a consequence, there is an increasing concern among users regarding the reliability and availability of Cloud computing services. Fault tolerance and resilience serve as an effective means to address user's reliability and availability concerns. In this chapter, we focus on characterizing the recurrent failures in a typical Cloud computing environment, analyzing the effects of failures on user's applications, and surveying fault tolerance solutions corresponding to each class of failures. We also discuss the perspective of offering fault tolerance as a service to user's applications as one of the effective means to address user's reliability and availability concerns.

Fault Tolerance and Resilience in Cloud Computing Environments

Ravi Jhawar and Vincenzo Piuri

1. Introduction

Cloud computing is gaining an increasing popularity over traditional information processing systems. Service providers have been building massive data centers that are distributed over several geographical regions to efficiently meet the demand for their Cloud-based services (e.g., [AWS], [Azure], [GCP]). In general, these data centers are built using hundreds of thousands of commodity servers, and virtualization technology is used to provision computing resources (e.g., by delivering Virtual Machines – VMs – with a given amount of CPU, memory and storage capacity) over the Internet by following the pay-per-use business model (e.g., [AWS.EC2]). Leveraging the economies of scale, a single physical host is often used as a set of several virtual hosts by the service provider, and benefits such as the semblance of an inexhaustible set of available computing resources is provided to the users. As a consequence, an increasing number of users are moving to Cloud-based services for realizing their applications and business processes.

The use of commodity components however exposes the hardware to conditions that it was not originally designed for [FRM.2012, VN.2010]. Moreover, due to the highly complex nature of the underlying infrastructure, even carefully engineered data centers are subject to a large number of failures [HB.2009]. These failures evidently reduce the overall reliability and availability of the Cloud computing service. As a result, fault tolerance becomes of paramount importance to the users as well as the service providers to ensure correct and continuous system operation even in the presence of an unknown and unpredictable number of failures.

The dimension of risks on the user's applications deployed in the virtual machine instances in a Cloud has also changed since the failures in data centers are normally outside the scope of the user's organization. Moreover, traditional ways of achieving fault tolerance require users to have an in-depth knowledge of the underlying mechanisms, whereas, due to the abstraction layers and business model of Cloud computing, system's architectural details are not widely available to the users. This implies that traditional methods of introducing fault tolerance may not be very effective in the Cloud computing context and there is an increasing need to address user's reliability and availability concerns.

Goal of this chapter is to develop an understanding on the nature, numbers, and kind of faults that appear in typical Cloud computing infrastructures, how these faults impact user's applications, and how faults can be handled in an efficient and cost-effective manner. To this aim, we first describe the fault model of typical Cloud computing environments in Section 2 on

the basis of system architecture, failure characteristics of widely used server and network components, and analytical models. An overall understanding on the fault model may help researchers and developers to build more reliable Cloud computing services. We introduce some basic and general concepts on fault tolerance and summarize the parameters that must be taken into account when building a fault tolerant system in Section 3. A scheme in which different levels of fault tolerance can be achieved by user's applications by exploiting the properties of the Cloud computing architecture is then presented in Section 4.

In Section 5 we discuss a solution that can function on user's applications in a general and transparent manner to tolerate one of the two most frequent classes of faults that appear in the Cloud computing environment. In section 6 we present a scheme that can tolerate the other class of frequent faults while reducing the overall resource costs by half when compared to existing solutions in the literature. These two techniques, along with the concept of different fault tolerance levels, are used as the basis to develop a methodology and framework that offers fault tolerance as an additional service to the user's applications (see Section 7). We believe that the notion of offering fault tolerance as a service may serve as an efficient alternative to traditional approaches in addressing user's reliability and availability concerns.

2. Cloud computing fault model

In general, a failure represents the condition in which the system deviates from fulfilling its intended functionality or the expected behavior. A failure happens due to an error; that is, due to reaching an invalid system state. The hypothesized cause for an error is a fault which represents a fundamental impairment in the system. The notion of faults, errors and failures can be represented using the following chain [S.2004, HH.1997]:

... Fault → Error → Failure → Fault → Error → Failure ...

Fault tolerance is the ability of the system to perform its function even in the presence of failures. This implies that it is utmost important to clearly understand and define what constitutes the correct system behavior so that specifications on its failure characteristics can be provided and consequently a fault tolerant system be developed. In this section, we discuss the fault model of typical Cloud computing environments to develop an understanding on the numbers as well as the causes behind recurrent system failures. In order to analyze the distribution and impact of faults, we first describe the generic Cloud computing architecture.

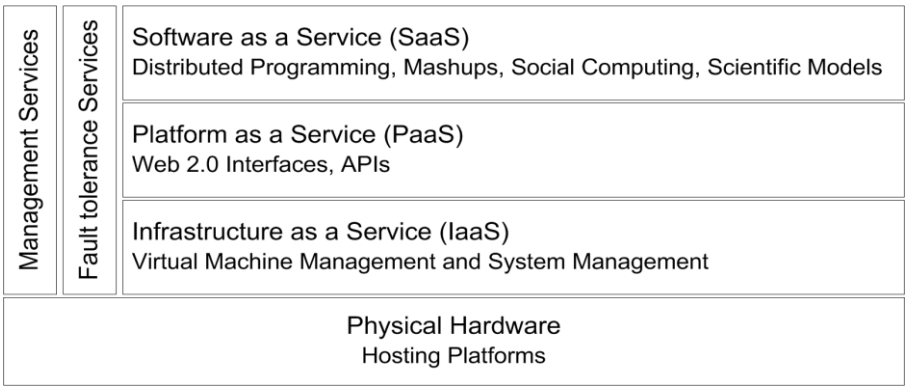

Fig. 1 Layered Architecture of Cloud computing

2.1. Cloud computing architecture

Cloud computing architecture comprises four distinct layers as illustrated in Figure 1 [AFG.2009]. Physical resources (e.g., blade servers and network switches) are considered as the lowest-layer in the stack, on top which, virtualization and system management tools are embedded to form the Infrastructure-as-a-Service (IaaS) layer. Note that the infrastructure supporting large-scale Cloud deployments is typically the data centers and virtualization technology is used to maximize the use of physical resources, application isolation and quality of service. Services offered by IaaS are normally accessed through a set of user-level middleware services which provide an environment to simplify application development and deployment (e.g., web 2.0 interfaces, libraries and programming languages). The layer above the IaaS which binds all user-level middleware tools is referred as Platform-as-a-Service (PaaS). User-level applications (e.g., social networks and scientific models) that are built and hosted on top of the PaaS layer comprise the Software-as-a-Service (SaaS) layer.

Failure in a given layer normally has an impact on the services offered by the layers above it. For example, failure in a user-level middleware (PaaS) may produce errors in the software services built on top of it (SaaS applications). Similarly, failures in the physical hardware or the IaaS layer will have an impact on most PaaS and SaaS services. This implies that the impact of failures in the IaaS layer or the physical hardware is significantly high; hence, it is important to characterize typical hardware faults and develop corresponding fault tolerance techniques.

We describe the failure behavior of various server components based on the statistical information obtained from large-scale studies on data center failures using data mining techniques [VN.2010, GJN.2011] and analyze the impact of component failures on user's applications by means of analytical models such as fault trees and Markov chains [JP.2012].

Similarly to server components, we also present the failure behavior of network component failures.

2.2. Failure behavior of servers

Each server in the data center typically contains multiple processors, storage disks, memory modules and network interfaces. The study about server failure and hardware repair behavior is to be performed using a large collection of servers (approximately 100,000 servers) and corresponding data on part replacement such as details about server configuration, when a hard disk was issued a ticked for replacement and when it was actually replaced. Such data repository which included server collection spanning multiple data centers distributed across different countries is gathered and inferred in [VN.2010]. Key observations derived from this study are as follows:

- 92% of the machines do not see any repair events but the average number of repairs for the remaining 8% is 2 per machine (20 repair/replacement events contained in 9 machines were identified over a 14 months period). The annual failure rate (AFR) is therefore around 8%.
- For an 8% AFR, repair costs that amount to 2.5 million dollars are approximately spent for 100,000 servers.
- About 78% of total faults/replacements were detected on hard disks, 5% on RAID controllers and 3% due to memory failures. 13% of replacements were due to a collection of components (not particularly dominated by a single component failure). Hard disks are clearly the most failure-prone hardware components and the most significant reason behind server failures.
- About 5% of servers experience a disk failure in less than 1 year from the date when it is commissioned (young servers), 12% when the machines are 1 year old, and 25% of the servers sees hard disk failures when it is 2 years old.
- Interestingly, based on the Chi-squared automatic interaction detector methodology, none of the following factors: age of the server, its configuration, location within the rack and workload run on the machine were found to be a significant indicator for failures.
- Comparison between the number of repairs per machine (RPM) against the number of disks per server in a group of servers (clusters) indicates that (i) there is a relationship in the failure characteristics of servers that have already experienced a failure, and (ii) the number of RPM has a correspondence to the total number of disks on that machine.

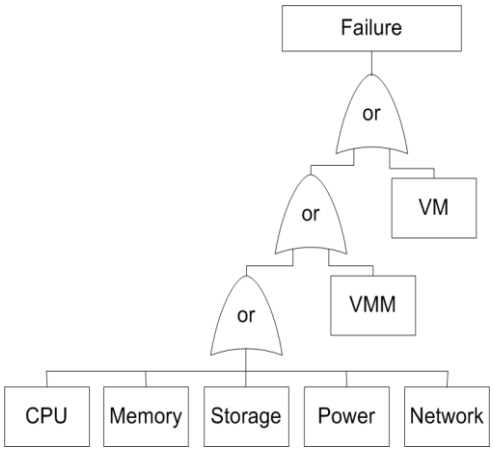

Fig. 2a Fault tree characterizing server failures [JP.2012]

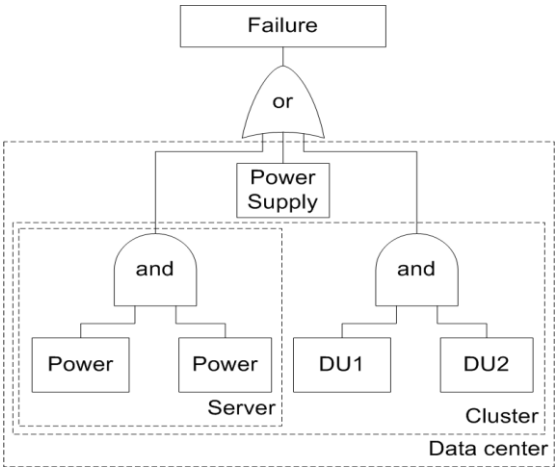

Fig. 2b Fault tree characterizing power failures [JP.2012]

Based on these statistics, it can be inferred that robust fault tolerance mechanisms must be applied to improve the reliability of hard disks (assuming independent component failures) to substantially reduce the number of failures. Furthermore, to meet the high availability and reliability requirements, applications must reduce utilization of hard disks that have already experienced a failure (since the probability of seeing another failure on that hard disk is higher).

Failure behavior of servers can also be analyzed based on the models defined using fault trees and Markov chains [JP.2012, STT.2008]. The rationale behind the modeling is twofold: (i) to capture the user's perspective on component failures, that is, understand the behavior of user's applications that are deployed in the VM instances under server component failures and (ii) to define the correlation between individual component failures and the boundaries on the impact of each failure. An application may have an impact when there is a failure/error either in the processor, memory modules, storage disks, power supply or network interfaces of the server, or the hypervisor, or the VM instance itself. Figure 2a illustrates this behavior as a fault tree where the top-event represents a failure in the user's application. Reliability and availability of each server component must be derived using Markov models that are populated using long-term failure behavior information such as the one described in [VN.2010].

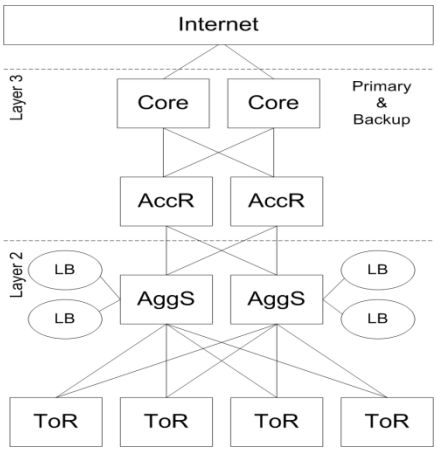

Fig. 3a Partial network architecture of a data center [GJN.2011]

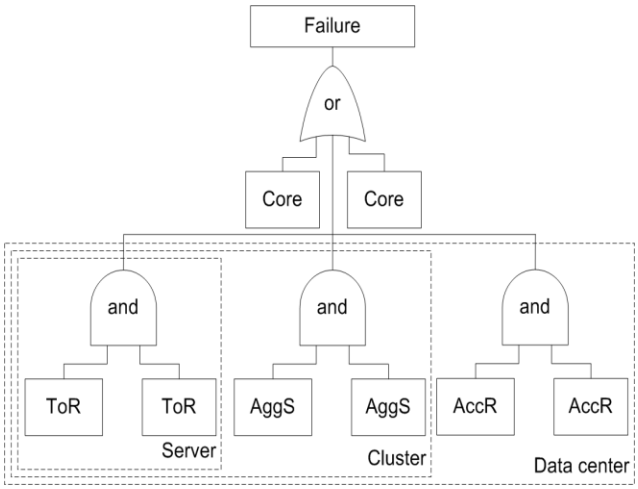

Fig. 3b Fault tree characterizing network failures [JP.2012]

2.3. Failure behavior of the network

It is important to understand the overall network topology and various network components involved in constructing a data center so as to characterize the network failure behavior. Figure 3a illustrates an example of partial data center network architecture [Cisco.2004, GJN.2011]. Servers are connected using a set of network switches and routers. In particular, all rack-mounted servers are first connected via a 1Gbps link to a top-of-rack switch (ToR), which is in turn connected to two (primary and backup) aggregation switches (AggS). An AggS connects tens of switches (ToR) to redundant access routers (AccR). This implies that each AccR handles traffic from thousands of servers and route it to core routers that connect different data centers to the Internet [JP.2012, GJN.2011]. All links in the data centers commonly use Ethernet as the link layer protocol and redundancy is applied to all network components at each layer in the network topology (except for ToRs). In addition, redundant pairs of load balancers (LBs) are connected to each AggS and mapping between static IP address presented to the users and dynamic IP addresses of internal servers that process user's requests is performed. Similarly to the study on failure behavior of servers, a large scale study on the network failures in data centers is performed in [GJN.2011]. A link failure happens when the connection between two devices on a specific interface is down and a device failure happens when the device is not routing/forwarding packets correctly (e.g., due to power outage or hardware crash). Key observations derived from this study are as follows:

- Among all the network devices, load balancers are least reliable (with failure probability of 1 in 5) and ToRs are most reliable (with a failure rate of less than 5%). The root causes for failures in LBs are mainly the software bugs and configuration

errors (as opposed to the hardware errors for other devices). Moreover, LBs tend to experience short but frequent failures. This observation indicates that low-cost commodity switches (e.g., ToRs and AggS) provide sufficient reliability.

- The links forwarding traffic from LBs have highest failure rates; links higher in the topology (e.g., connecting AccRs) and links connecting redundant devices have second highest failure rates.
- The estimated median number of packets lost during a failure is 59K and median number of bytes is 25MB (average size of lost packets is 423Bytes). Based on prior measurement studies (that observe packet sizes to be bimodal with modes around 200Bytes and 1,400Bytes), it is estimated that most lost packets belong to the lower part (e.g., ping messages or ACKs).
- Network redundancy reduces the median impact of failures (in terms of number of lost bytes) by only 40%. This observation is against the common belief that network redundancy completely masks failures from applications.

Therefore, the overall data center network reliability is about 99.99% for 80% of the links and 60% of the devices. Similar to servers, Figure 3b represents the fault tree for user's application failure with respect to network failures in the data center. A failure happens when there is an error in all redundant switches ToRs, AggS, AccR or core routers, or the network links connecting physical hosts. Since the model is designed in the user's perspective, a failure in this context implies that the application is not connected to the rest of the network or gives errors during data transmission. Using this modeling technique, the boundaries on the impact of each network failure can be represented (using server, cluster and data center level blocks) and further be used to increase the fault tolerance of user's application (e.g., by placing replicas of an application in different failure zones).

3. Basic concepts on fault tolerance

In general, the faults we analyzed in Section 2 can be classified in different ways depending on the nature of the system. Since, in this chapter, we are interested in typical Cloud computing environment faults that appear as failures to the end users, we classify the faults into two types similarly to other distributed systems:

- *Crash faults* that cause the system components to completely stop functioning or remain inactive during failures (e.g., power outage, hard disk crash)
- *Byzantine faults* that leads the system components to behave arbitrarily or maliciously during failure, causing the system to behave unpredictably incorrect.

As reminded previously, fault tolerance is the ability of the system to perform its function even in the presence of failures. It serves as one of the means to improve the overall system's

dependability. In particular, it contributes significantly in increasing system's reliability and availability.

The most widely adopted methods to achieve fault tolerance against crash faults and byzantine faults are as follows:

- *Checking and monitoring*: The system is constantly monitored at runtime to validate, verify and ensure that correct system specifications are being met. This technique, while very simple, plays a key role in failure detection and subsequent reconfiguration.
- *Checkpoint and restart*: The system state is captured and saved based on pre-defined parameters (e.g., after every 1024 instructions or every 60 seconds). When the system undergoes a failure, it is restored to the previously known correct state using the latest checkpoint information (instead of restarting the system from start).
- *Replication*: Critical system components are duplicated using additional hardware, software and network resources in such a way that a copy of the critical components is available even after a failure happens. Replication mechanisms are mainly used in two formats: active and passive. In active replication, all the replicas are simultaneously invoked and each replica processes the same request at the same time. This implies that all the replicas have the same system state at any given point of time (unless designed to function in an asynchronous manner) and it can continue to deliver its service even in case of a single replica failure. In passive replication, only one processing unit (the primary replica) processes the requests while the backup replicas only save the system state during normal execution periods. Backup replicas take over the execution process only when the primary replica fails.

Variants of traditional replication mechanisms (active and passive) are often applied on modern distributed systems. For example, semi-active replication technique is derived from traditional approaches wherein primary and backup replicas execute all the instructions but only the output generated by the primary replica is made available to the user. Output generated by the backup replicas is logged and suppressed within the system so that it can readily resume the execution process when the primary replica failure happens. Figure 4a depicts the Markov model of a system that uses active/semi-active replication scheme with two replicas [JP.2012]. This model serves as an effective means to derive the reliability and availability of the system because failure behavior of both replicas can be taken into account. Moreover, as described in Section 2, the results of the Markov model analysis can be used to support the fault trees in characterizing the impact of failures in the system. Each state in the model is represented by a pair (x, y) where x=1 denotes that the primary replica is working and x=0 implies that it failed. Similarly, y represents the working condition of the backup replica. The system starts and remains in state (1,1) during normal execution, i.e., when both the replicas are available and working correctly. A failure either in the primary or the backup

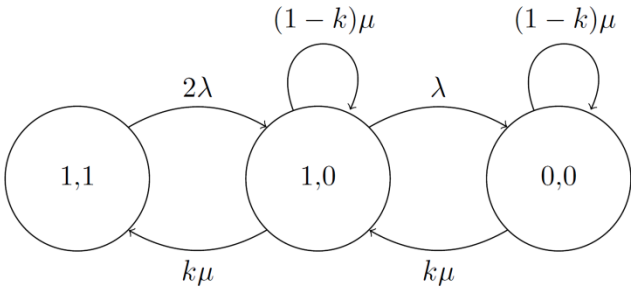

Fig. 4a Markov model of a system with two replicas in Active/Semi-active replication scheme [JP.2012]

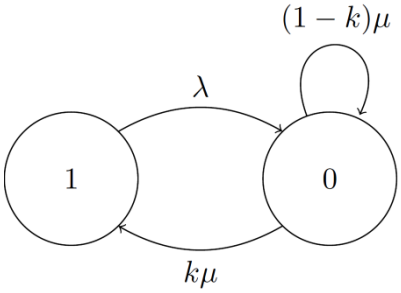

Fig. 4b Markov model of a system with two replicas in Passive replication scheme [JP.2012]

replica moves the system to state (0,1) or (1,0) where the other replica takes over the execution process. A single state is sufficient to represent this condition in the model since both replicas are consistent with each other. The system typically initiates its recovery mechanism in state (0,1) or (1,0), and moves to state (1,1) if the recovery of failed replica is successful; otherwise it transits to state (0,0) and becomes completely unavailable. Similarly, Figure 4b illustrates the Markov model of the system for which a passive replication scheme is applied. λ denotes the failure rate, μ denotes the recovery rate and k is a constant.

Fault tolerance mechanisms are varyingly successful in tolerating faults [ABL.2008]. For example, a passively replicated system can tolerate only crash faults whereas actively replicated system using $3f$+1 replicas are capable of tolerating byzantine faults. In general, mechanisms that handle failures at a finer granularity, offering higher performance guarantees, also consume higher amount of resources [JPS.2012a]. Therefore, the design of fault tolerance mechanisms must take into account a number of factors such as implementation complexity, resource costs, resilience, and performance metrics, and achieve a fine balance of the following parameters:

- *Fault tolerance model*: Measures the strength of the fault tolerance mechanism in terms of the granularity at which it can handle errors and failures in the system. This factor is characterized by the robustness of failure detection protocols, state synchronization methods, and strength of the fail-over granularity.
- *Resource consumption*: Measures the amount and cost of resources that are required to realize a fault tolerance mechanism. This factor is normally inherent with the granularity of the failure detection and recovery mechanisms in terms of CPU, memory, bandwidth, I/O, and so on.
- *Performance*: This factor deals with the impact of the fault tolerance procedure on the end-to-end quality of service (QoS) both during failure and failure-free periods. This impact is often characterized using fault detection latency, replica launch latency and failure recovery latency, and other application-dependent metrics such as bandwidth, latency, and loss rate.

We build on the basic concepts discussed in this section to analyze the fault tolerance properties of various schemes designed for Cloud computing environment.

4. Different levels of fault tolerance in Cloud computing

As discussed in Section 2, server components in a Cloud computing environment are subject to failures, affecting user's applications, and each failure has an impact within a given boundary in the system. For example, a crash in the pair of aggregate switches may result in the loss of communication among all the servers in a cluster; in this context, the boundary of failure is the cluster since applications in other clusters can continue functioning normally. Therefore, while applying a fault tolerance mechanism such as a replication scheme, at least one replica of the application must be placed in a different cluster to ensure that aggregate switch failure does not result in a complete failure of the application. Furthermore, this implies that deployment scenarios (i.e., location of each replica) are critical to correctly realize the fault tolerance mechanisms. In this section, we discuss possible deployment scenarios in a Cloud computing infrastructure, and the advantages and limitations of each scenario.

Based on the architecture of the Cloud computing infrastructure, different levels of failure independence can be derived for Cloud computing services [GY.2010, UCH.2011]. Moreover, assuming that the failures in individual resource components are independent of each other, fault tolerance and resource costs of an application can be balanced based on the location of its replicas. Possible deployment scenarios and their properties are as follows.

- *Multiple machines within the same cluster*. Two replicas of an application can be placed on the hosts that are connected by a ToR switch i.e., within a LAN. Replicas deployed in this configuration can benefit in terms of low latency and high bandwidth but obtain very limited failure independence. A single switch or power distribution failure may result in an outage of the entire application and both replicas cannot communicate to complete the fault tolerance protocol. Cluster level blocks in the fault trees of each resource component (e.g., network failures as shown in Figure 3b) must be combined using a logical AND operator to analyze the overall impact of failures in the system. Note that reliability and availability values for each fault tolerance mechanism with respect to server faults must be calculated using a Markov model.
- *Multiple clusters within a data center*. Two replicas of an application can be placed on the hosts belonging to different clusters in the same data center i.e., on the hosts that are connected via a ToR switch and AggS. Failure independence of the application in this deployment context remains moderate since the replicas are not bound to an outage with a single power distribution or switch failure. The overall availability of an

application can be calculated using cluster level blocks from fault trees combined with a logical OR operator in conjunction with power and network using AND operator.

- *Multiple data centers*. Two replicas of an application can be placed on the hosts belonging to different data centers i.e., connected via a switch, AggS and AccR. This deployment has a drawback with respect to high latency and low bandwidth, but offers a very high level of failure independence. A single power failure has least effect on the availability of the application. The data center level blocks from the fault trees may be connected with a logical OR operator in conjunction with the network in the AND logic.

As an example, using the data published in [STT.2008, KMT.2009], overall availability of each representative replication scheme with respect to different deployment levels is obtained as shown in Table 1. Availability of the system is highest when the replicas are placed in two different data centers. The value reduces when replicas are placed in two different clusters within the same data center and it is lowest when replicas are placed inside the same LAN. The overall availability obtained by semi-active replication is higher than semi-passive replication, and lowest for simple passive replication scheme.

Table 1 Availability values (normalized to 1) for replication techniques at different deployment scenarios [JP.2012]

	Same Cluster	Same Data center, diff. clusters	Diff. Data centers
Semi-Active	0.9871	0.9913	0.9985
Semi-Passive	0.9826	0.9840	0.9912
Passive	0.9542	0.9723	0.9766

As described in Section 3, effective implementation of fault tolerance mechanisms requires consideration of the strength of fault tolerance model, resource costs and performance. While traditional fault tolerance methods require tailoring of each application having an in-depth knowledge of the underlying infrastructure, in Cloud computing scenario, it would also be beneficial to develop methodologies that can generically function on user's applications so that a large number of applications can be protected using the same protocol. In addition to generality, agility in managing replicas and checkpoints to improve the performance, and reduction in the resource consumption costs while not limiting the strength of fault tolerance mechanisms is required.

Although several fault tolerance approaches are being proposed for Cloud computing services, most solutions that achieve at least one of the required properties described above, are based on the virtualization technology. By using virtualization-based approaches, it is also possible to deal with both classes of faults that are discussed in Section 3. In particular, in Section 5 we present a virtualization-based solution that provides fault tolerance against crash failures using

a checkpointing mechanism. We discuss this solution because it offers two additional, significantly useful, properties: (i) fault tolerance is induced independent to the applications and hardware on which it runs. In other words, an increased level of generality is achieved since any application can be protected using the same protocol as long as it is deployed in a VM, and (ii) mechanisms such as replication, failure detection and recovery are applied transparently -- not modifying the OS or application's source code. Then, in Section 6 we present a virtualization-based solution that uses typical properties of a Cloud computing environment to tolerate byzantine faults using a combination of replication and checkpointing techniques. We discuss this solution because it reduces the resource consumption costs incurred by typical byzantine fault tolerance schemes during fail-free periods nearly by half.

5. Fault tolerance against crash failures in Cloud computing

A scheme that leverages the virtualization technology to tolerate crash faults in the Cloud in a transparent manner is discussed in this section. The system or user application that must be protected from failures is first encapsulated in a VM (say active VM or the primary), and operations are performed at the VM level (in contrast to traditional approach of operating at the application level) to obtain paired servers that run in active-passive configuration. Since the protocol is applied at the VM level, this scheme can be used independent of the application and underlying hardware, offering an increased level of generality. In particular, we discuss the design of *Remus* as an example system that offers the above mentioned properties [CLM.2008]. Remus aims to provide high availability to the applications, and to achieve this, it works in four phases:

1. Checkpoint the changed memory state at the primary and continue to next epoch of network and disk request streams.
2. Replicate system state on the backup.
3. Send checkpoint acknowledgement from the backup when complete memory checkpoint and corresponding disk requests have been received.
4. Release outbound network packets queued during the previous epoch upon receiving the acknowledgement.

Remus achieves high-availability by frequently checkpointing and transmitting the state of the active VM on to a backup physical host. The VM image on the backup is resident in the memory and may begin execution immediately after a failure in the active VM is detected. The backup only acts like a receptor since the VM in the backup host is not actually executed during fail-free periods. This allows the backup to concurrently receive checkpoints from VMs running on multiple physical hosts (in an N-to-1 style configuration) providing a higher degree of freedom in balancing resource costs due to redundancy.

In addition to generality and transparency, seamless failure recovery can be achieved i.e., no externally visible state is lost in case of a single host failure and recovery happens rapidly enough that it appears only like a temporary packet loss. Since the backup is only periodically consistent with the primary replica using the checkpoint-transmission procedure, all network output is buffered until a consistent image of the host is received by the backup, and the buffer is released only when the backup is completely synchronized with the primary. Unlike network traffic, the disk state is not externally visible but it has to be transmitted to the backup as part of a complete cycle. To address this, Remus asynchronously sends the disk state to the backup where it is initially buffered in the RAM. When the corresponding memory state is received, complete checkpoint is acknowledged, output is made visible to the user, and buffered disk state is applied to the backup disk.

Remus is built on Xen hypervisor's live migration machinery [CFH.2005]. Live migration is a technique using which a complete VM can be relocated onto another physical host in the network (typically a LAN) with a minor interruption to the VM. Xen provides an ability to track guest's writes to memory using a technique called shadow page tables. During live migration, memory of the VM is copied to the new location while the VM still continues to run normally at the old location. The writes to the memory are then tracked and the dirtied pages are transferred to the new location periodically. After a sufficient number of iterations, or when no progress in copying the memory is being made (i.e., when the VM is writing to the memory as fast as the migration process), the guest VM is suspended, remaining dirtied memory along with the CPU state is copied and the VM image in the new location is activated. The total migration time depends on the amount of dirtied memory during guest execution, and total downtime depends on the amount of memory remaining to be copied when the guest is suspended. The protocol design of the system, particularly each checkpoint, can be viewed as the final stop-and-copy phase of live-migration. The guest memory in live migration is iteratively copied incurring several minutes of execution time. The singular stop-and-copy (the final step) operation incurs a very limited overhead – typically in the order of a few milliseconds.

While Remus provides an efficient replication mechanism, it employs a simple failure detection technique that is directly integrated within the checkpoint stream. A timeout of the backup in responding to commit requests made by the primary will result in the primary suspecting a failure (crash and disabled protection) in the backup. Similarly, a timeout of the new checkpoints being transmitted from the primary will result in the backup assuming a failure in the primary. At this point, the backup begins execution from the latest checkpoint.

The protocol is evaluated (i) to understand whether or not the overall approach is practically deployable and (ii) to analyze the kind of workloads that are most amenable to this approach.

Correctness evaluation is performed by deliberatively injecting network failures at each phase of the protocol. The application (or the protected system) runs a kernel compilation process to generate CPU, memory and disk load, and a graphics intensive client (glxgears) attached to X11 server is simultaneously executed to generate the network traffic. Checkpoint frequency is configured to 25 milliseconds and each test is performed two times. It is reported that the backup successfully took over the execution for each failure with a network delay of about 1 second when the backup detected the failure and activated the replicated system. The kernel compilation task continued to completion and glxgears client resumed after a brief pause. The disk image showed no inconsistencies when the VM was gracefully shut down.

Performance evaluation is performed using the SPECweb benchmark that is composed of a web server, an application server and one or more web client simulators. Each tier (server) was deployed in a different VM. The observed scores decrease up to 5 times the native score (305) when the checkpointing system is active. This behavior is mainly due to network buffering; the observed scores are much higher when network buffering is disabled. Furthermore, it is reported that at configuration rates of 10, 20, 30 and 40 checkpoints per second, the average checkpoint rate achieved are 9.98, 16.38, 20.25 and 23.34 respectively. This behavior can be explained with SPECweb's very fast memory dirtying resulting in slower checkpoints than desired. The realistic workload hence illustrates that the amount of network traffic generated by the checkpointing protocol is considerably large, and as consequence, this system is not well suited for applications that are very sensitive to network latencies.

Therefore, virtualization technology can largely be exploited to develop general-purpose fault tolerance schemes that can be applied to handle crash faults in a transparent manner.

6. Fault tolerance against byzantine failures in Cloud computing

Byzantine Fault Tolerance (BFT) protocols are powerful approaches to obtain highly reliable and available systems. Despite numerous efforts, most BFT systems have been too expensive for practical use – so far, no commercial data centers have employed BFT techniques. For example, the BFT algorithm presented in [CL.1999] for asynchronous, distributed, client-server systems requires at least $3f+1$ replica (1 primary and remaining backup) to execute a three-phase protocol that can tolerate f byzantine faults. Note that, as described in Section 3, systems that tolerate faults at a finer granularity such as the byzantine faults also consume very high amounts of resources, and as discussed in Section 4, it is critical to consider the resource costs while implementing a fault tolerance solution.

The high resource consumption cost in BFT protocols is most likely due to the way faults are normally handled. BFT approaches typically replicate the server (state machine replication -- SMR) and each replica is forced to execute the same request in the same order. This

enforcement requirement demands the server replicas to reach an agreement on the ordering of a given set of requests even in the presence of byzantine faulty servers and clients. For this purpose, an agreement protocol that is referred as *Byzantine Agreement* is used. When an agreement on the ordering is reached, service execution is performed and majority voting scheme is devised to choose the correct output (and to detect the faulty server). This implies that two clusters of replicas are necessary to realize BFT protocols.

It is observed that when realistic data center services implement BFT protocols, the dominant costs are due to the hardware performing service execution and not due to running the agreement protocol [WSV.2011]. For instance, a toy application running *null* requests with the Zyzzyva BFT approach [KAD.2009] exhibits a peak throughput of 80K requests/second while a database service running the same protocol on comparable hardware exhibits almost three times lower throughput. Based on this observation, ZZ, an execution approach that can be integrated with existing BFT SMR and agreement protocols is presented in [WSV.2011]. The prototype of ZZ is built on the BASE implementation [CL.1999] and guarantees BFT while significantly reducing resource consumption costs during fail-free periods. Table 2 compares resource costs of well-known BFT techniques. Since ZZ provides an effective balance between resource consumption costs and fault tolerance model, further in this section we discuss its system design in detail.

Table 2 Resource consumption costs incurred by well-known byzantine fault tolerance protocols [WSV.2011]

	PBFT [CL.1999]	SEP [YMV.2003]	Zyzzyva [KAD.2009]	ZZ [WSV.2011]
Agreement replicas	$3f$+1	$3f$+1	$3f$+1	$3f$+1
Execution replicas	$3f$+1	$2f$+1	$2f$+1	(1+r)f+1

The design of ZZ is based on the virtualization technology and targeted to tolerate byzantine faults while reducing the resource provisioning costs incurred by BFT protocols during fail-free periods. The cost reduction benefits of ZZ can be obtained only when BFT is used in the data center running multiple applications so that sleeping replicas can be distributed across the pool of servers and higher peak throughput can be achieved when execution dominates the request processing cost and resources are constrained. These assumptions make ZZ a suitable scheme to be applied in a Cloud computing environment.

The system model of ZZ makes the following assumptions similar to most existing BFT systems:

- The service is either deterministic or non-deterministic operations in the service can be transformed to deterministic ones using an agreement protocol (i.e., ZZ assumes a SMR based BFT system).
- The system involves two kinds of replicas (i) *agreement replicas* that assign an order to client's requests and (ii) *execution replicas* that execute each client's request in the same order and maintain the application state.
- Each replica fails independently and exhibits Byzantine behavior (i.e., faulty replicas and clients may behave arbitrarily).
- An adversary can coordinate faulty nodes in an arbitrary manner, but it cannot circumvent standard cryptographic measures (e.g., collision resistant hash functions, encryption scheme and digital signatures).
- An upper bound g on number of faulty agreement replicas and f execution replicas is assumed for a given window of vulnerability.
- System can ensure safety in an asynchronous network, but liveness is guaranteed only during periods of synchrony.

Since the system runs replicas inside virtual machines, to maintain failure independence, it requires that a physical host can deploy at most one agreement and one execution replicas of the service simultaneously. The novelty in the system model is that it considers a Byzantine hypervisor. Note that, as a consequence of the above replica placement constraint, a malicious hypervisor can be treated by simply considering a single fault in all the replicas deployed on that physical host. Similarly, an upper bound f on the number of faulty hypervisors is assumed.

The BFT execution protocol reduces the replication cost from $2f$+1 to f+1 based on the following principle:

- A system that is designed to function correctly in an asynchronous environment will provide correct results even if some of the replicas are outdated.
- A system that is designed to function correctly in the presence of f Byzantine faults will, during fault-free period, remain unaffected even if up to f replicas are turned off.

The second observation is used to commission only f+1 replica to actively execute requests. The system is in a correct state if the response obtained from all f+1 replica is the same. In case of a failure (i.e., when responses do not match), the first observation is used to continue system operation as if the f standby replicas were slow but correct replicas.

To correctly realize this design, the system requires an agile replica wake-up mechanism. To achieve this, the system exploits virtualization technology by maintaining additional replicas (VMs) in a "dormant" state, which are either pre-spawned but paused VMs or the VM that is hibernated to a disk. There is a trade-off in adopting either method. Pre-spawned VM can resume execution in very short span (in the order of few milliseconds) but consumes memory

higher resources, whereas, VMs hibernated to disks incur greater recovery times but occupy only storage space. This design also raises several interesting challenges such as *how can a restored replica obtain the necessary application state that is required to execute the current request? How can the replication cost be made robust to faulty replica or client behavior? Does the transfer of entire application state take unacceptably long time?*

The system builds on the BFT protocol that uses independent agreement and execution clusters (similar to [YMV.2003]). Let A represent the set of replicas in the agreement cluster, $|A| = 2g + 1$, that run the three-phase agreement protocol [CL.1999]. When a client c sends its request Q to the agreement cluster to process an operation o with timestamp t, the agreement cluster assigns a sequence number n to the request. The timestamp is used to ensure that each client request is executed only once and a faulty client behavior does not affect other clients' requests. When an agreement replica j learns of the sequence number n committed to Q, it sends a commit message C to all execution replicas.

Let E represent the set of replicas in the execution cluster where $|E| = f + 1$ during fail-free periods. When an execution replica i receives $2g + 1$ valid and matching commit messages from A, in the form of a commit certificate $\{C_i\}$, $i \in A|2g + 1$, and if it has already processed all the requests with lower sequence than n, it produces a reply R and sends it to the client. The execution cluster also generates an execution report ER for the agreement cluster.

During normal execution, the response certificate $\{R_i\}$, $i \in E|f + 1$ obtained by the client matches replies from all $f + 1$ execution nodes. To avoid unnecessary wakeups due to a partially faulty execution replica which replies correctly to the agreement cluster but delivers a wrong response to the client, ZZ introduces an additional check as follows: when the replies are not matching, the client resends the same request to the agreement cluster. The agreement cluster sends a reply affirmation RA to the client if it has $f + 1$ valid responses for the retransmitted request. In this context, the client accepts the reply if it receives $g + 1$ messages containing a response digest $\bar{R}$ that matches one of the replies already received. Finally, if the agreement cluster does not generate an affirmation for the client, additional nodes are started.

ZZ uses periodic checkpoints to update the state of newly commissioned replicas and to perform garbage collection on replica's logs. Execution nodes create checkpoints of the application state and reply logs, generate a checkpoint proof CP, and send it all execution and agreement nodes. The checkpoint proof is in the form of a digest that allows recovering node in identifying the checkpoint data they obtain from potentially faulty nodes, and the checkpoint certificate $\{CP_i\}$, $i \in E|f + 1$ is a set of $f + 1$ CP messages with matching digests.
Fault detection in the execution replicas is based on timeouts. Both lower and higher values of timeouts may impact the system's performance. The former may falsely detect failures and the later may provide a window to the faulty replicas to degrade the system's performance. To set

appropriate timeouts, ZZ suggests the following procedure: the agreement replica sets the timeout $\tau_n$to Kt_1 upon receiving the first response to the request with sequence number n; $t_1$is the response time and K is a preconfigured variance bound. Based on this trivial theory, ZZ proves that a replica faulty with a given probability p can inflate average response time by a factor of:

$$\max\left(1, \sum_{0 \le m \le f} P(m)I(m)\right)$$

where:

$$P(m) = \binom{f}{m} p^m (1-p)^{f-m}$$

$$I(m) = max\left(1, \frac{K.E[MIN_{f+1-m}]}{E[MAX_{f+1}]}\right)$$

$P(m)$ represents the probability of m simultaneous failures and $I(m)$ is the response time inflation that m faulty nodes can inflict. Assuming identically distributed response times for a given distribution, $E[MIN_{f+1-m}]$ is the expected minimum time for a set of $f+1-m$ replicas and $E[MAX_{f+1}]$ is the expected maximum response time of all $f+1$ replicas [WSV.2011]. Replication costs vary from $f+1$ to $2f+1$ depending on the probability of replicas being faulty p and the likelihood of false timeouts π_1. Formally, the expected replication cost is less than $(1+r)f+1$, where $r = 1-(1-p)^{f+1} + (1-p)^{f+1}\pi_1$.

Therefore, virtualization technology can be effectively used to realize byzantine fault tolerance mechanisms at a significantly lower resource consumption costs.

7. Fault tolerance as a service in Cloud computing

The drawback of the solutions discussed in Section 5 and Section 6 is that the user must either tailor its application using a specific protocol (e.g., ZZ) by taking into account the system architecture details, or require the service provider to implement a solution for its applications (e.g., Remus). Note that the (i) fault tolerance properties of the application remain constant throughout its life-cycle using this methodology and (ii) users may not have all the architectural details of service provider's system. However, the availability of a pool of fault tolerance mechanisms that provide transparency and generality can allow realization of the notion of fault tolerance as a service. The latter perspective to fault tolerance intuitively provides immense benefits.

As a motivating example, consider a user that offers a web-based e-commerce service to its customers that allows them to pay their bills and manage fund transfers over the Internet. The user implements the e-commerce service as a multi-tier application that uses the storage service of the service provider to store and retrieve its customer data, and compute service to process its operations and respond to customer queries. In this context, a failure in service provider's system can impact the reliability and availability of the e-commerce service. The implications of storage server failure may be much higher than a failure in one of the compute nodes. This implies that each tier of the e-commerce application must possess different levels of fault tolerance, and the reliability and availability goals may change over time based on the business demands. Using traditional methods, fault tolerance properties of the e-commerce application remains constant throughout its life-cycle and hence, in user's perspective, it is complementary to engage with a third party (the fault tolerance service provider ftSP), specify its requirements based on the business needs, and transparently possess desired fault tolerance properties without studying the low level fault tolerance mechanisms.

The ftSP must realize a range of fault tolerance techniques as individual modules (e.g., separate agreement and execution protocols, and heartbeat based fault detection technique as an independent module) to benefit from the economies of scale. For example, since failure detection technique in Remus and ZZ are based on the same principle, instead of integrating the liveness requests within the checkpointing stream, the heartbeat test module can be reused in both the solutions. However, realization of this notion requires a technique for selection of appropriate fault tolerance mechanisms based on user's requirements and a general purpose framework that can integrate with the Cloud computing environment. Without such a framework, individual applications must implement its own solution, resulting in highly complex system environment. Further in this section, we present a solution that supports ftSP to realize its service effectively.

In order to abstract low-level system procedures from the users, a new dimension to fault tolerance is presented in [JPS.2012b] wherein applications deployed in the VM instances in a Cloud computing environment can obtain desired fault tolerance properties from a third-party as a service. The new dimension realizes a range of fault tolerance mechanisms that can transparently function on user's applications as independent modules, and a set of metadata is associated with each module to characterize its fault tolerance properties. The metadata is used to select appropriate mechanisms based on user's requirements. A complete fault tolerance solution is then composed using selected fault tolerance modules and delivered to the user's application.

Consider ft_unit to be the fundamental module that applies a coherent fault tolerance mechanism, in a transparent manner, to a recurrent system failure at the granularity of a VM instance. An ft_unit handles the impact of hardware failures by applying fault tolerance

mechanisms at the virtualization layer rather than user's application. Examples of ft_units include the replication scheme for the e-commerce application that uses checkpointing technique such as Remus (ft_unit1), and node failures detection technique using heartbeat test (ft_sol2). Assuming that the ftSP realizes a range of fault tolerance mechanisms as ft_units, a two stage delivery scheme that can deliver fault tolerance as a service is as follows:

The *design stage* starts when a user requests the ftSP to deliver a solution with a given set of fault tolerance properties to its application. Each ft_unit provides a unique set of properties; the ftSP banks on this observation and defines the fault tolerance property p corresponding to each ft_unit as $p = (u, \hat{p}, A)$ where u represents the ft_unit, $\hat{p}$ denotes the high level abstract properties such as reliability and availability, and A denotes the set of functional, structural and operational attributes that characterizes the ft_unit u. The set A sufficiently refers to the granularity at which the ft_unit can handle failures, its limitations and advantages, resource consumption costs and quality of service parameters. Each attribute $a \in A$ takes a value $v(a)$ from a domain D_a and a partial (or total) ordered relationship is defined on the domainD_a. The values for the abstract properties are derived using the notion of fault trees and Markov model as described for the availability property in Table 1. An example of fault tolerance property for the ft_unit u_1 is $p = (u_1,\ \hat{p}$ =*{reliability=98.9%, availability=99.95%},* A =*{mechanism=semi-active_replication, fault_model=server_crashes, power_outage, number_of_replicas=4}*)

Similar to the domain of attribute values, a hierarchy of fault tolerance properties $\leq_p$ is also defined: if P is the set of properties, and given two properties $p_i, p_j \in P$, $p_i \leq_p p_j$ if $p_i.\hat{p} = p_j.\hat{p}$ and for all $a \in A$, $v_i(a) \leq v_j(a)$. This hierarchy suggests that all ft_units that hold the property p_j also satisfy the property p_i. Fault tolerance requirements of the users are assumed to be specified as desired properties p_c, and for each user request, the ftSP first generates a shortlisted set S of ft_units that match p_c. Each ft_unit within the set S is then compared, and an ordered list based on user's requirements is created. An example of the matching, comparison and selection process is as follows:

As an example, assume that the ftSP realizes three ft_units with properties
$\mathrm{p}_1 = (\mathrm{u}_1,\ \mathrm{A} =$ {mechanism=heartbeat_test, timeout_period=50ms, number_of_replicas=3, fault_model=node_crashes})
$\mathrm{p}_2 = (\mathrm{u}_2,\ \mathrm{A} =$ {mechanism=majority_voting, fault_model=programming_errors})
$\mathrm{p}_3 = (\mathrm{u}_3,\ \mathrm{A} =$ {mechanism=heartbeat_test, timeout_period=25ms, number_of_replicas=5, fault_model=node_crashes})
respectively. If the user requests fault tolerance support with a robust crash failure detection scheme, the set $\mathrm{S} = (\mathrm{u}_1, \mathrm{u}_3)$ is first generated (u_2 is not included in the set because it doesn't target server crash failures alone, and its attribute values that contribute to robustness are not defined) and finally after comparing each ft_unit within S, ftSP leverages u_3 since it is more robust than u_1 .

Note that each ft_unit serves only as a single fundamental fault tolerance module. This implies that the overall solution ft_sol that must be delivered to the user's application can be obtained by combining a set of ft_units as per specific execution logic. For instance, a heartbeat test based fault detection module must be applied only after performing replication, and recovery mechanism must be applied after a failure is detected. In other words, ft_units must be used to realize as a process that provides a complete fault tolerance solution, such as:

ft sol[
invoke:ft unit(*VM-instances replication*)
invoke:ft unit(*failure detection*)
do{
execute(*failure detection ft unit*)
}while(*no failures*)
if(*failure detected*)
invoke:ft unit(*recovery mechanism*)
]

By composing ft_sol using a set of modules on-the-fly, the dimension and intensity of the fault tolerance support can be changed dynamically. For example, the more robust fault detection mechanism can be replaced with a less robust one in the ft_sol based on the user's business demands. Similarly, by realizing each ft_unit as a configurable module, resource consumption costs can also be made limited. For example, a replication scheme using 5 replicas can be replaced with one having 3 replicas if desired by the user.

The *runtime stage* starts immediately after ft_sol is delivered to the user. This stage is essential to maintain a high level of service because the context of the Cloud computing environment may change at runtime resulting in mutable behavior of the attributes. To this aim, the ftSP defines a set of rules R over attributes $a \in A$ and their values $v(a)$ such that the validity of all the rules $r \in R$ establishes that the property p is supported by ft_sol (violation of a rule indicates that the property is not satisfied). Therefore, in this stage, the attribute values of each ft_sol delivered to user's applications is continuously monitored at runtime and corresponding set of rules are verified using a validation function $f(s, R)$. The function returns true if all the rules are satisfied, otherwise, it returns false. The matching and comparison process defined for the design stage is used to generate a new ft_sol in case of a rule violation. By continuously monitoring and updating the attribute values, note that the fault tolerance service offers support valid throughout the life cycle of the application (both initially during design time and runtime).

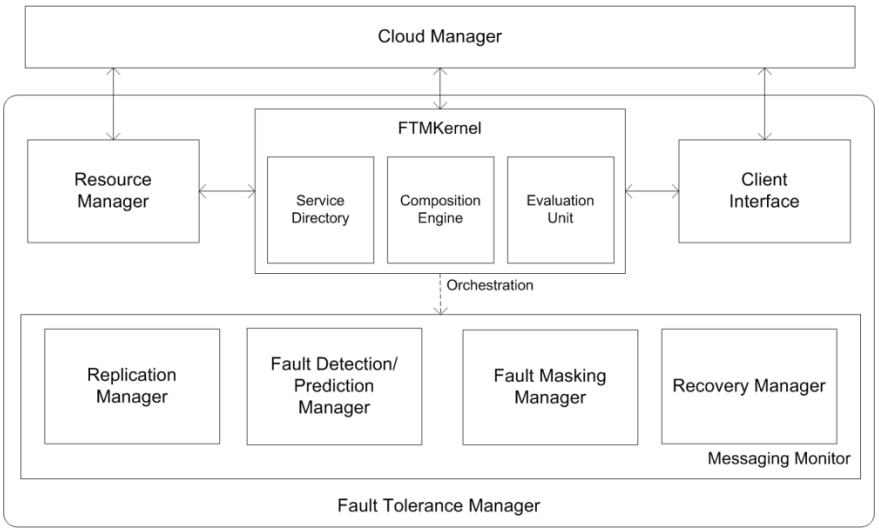

Fig. 5 Architecture of the Fault Tolerance Manager showing all the components

As an example, for a comprehensive fault tolerance solution ft_sol s_1 with property $p_1 = (s_1, \hat{p} = \{\text{reliability=98.9\%, availability=99.95\%}\}, A = \{\text{mechanism=active\_replication, fault\_detection=heartbeat\_test, number\_of\_replicas=4, recovery\_time=25ms}\})$, a set of rules R that can sufficiently test the validity of p_1 can be defined as:

r_1:number_of_server_instances ≥ 3
r_2:heartbeat_frequency $= 5$ms
r_3:recovery_time ≤ 25ms

These rules ensure that end reliability and availability is always greater than or equal to 98.9% and 99.95% respectively.

A conceptual architectural framework, the *Fault Tolerance Manager* (FTM), is also introduced in [JPS.2012] that provides the basis to realize the design stage and runtime stage of the delivery scheme, and serves as the basis to offer fault tolerance as a service. FTM is inserted as a dedicated service layer between the physical hardware and user applications along the virtualization layer. FTM is build using the principles of service oriented architectures, where each ft_unit is realized as an individual web service and ft_sol is created by orchestrating a set of ft_units (web services) using the business process execution language (BPEL) constructs. This allows the ftSP to satisfy its scalability and interoperability goals.

The central computing component, denoted as the FTMKernel, is composed of three main components:

- *Service Directory*: It is the registry of all ft_units realized by the service provider in the form of web services that i) describes its operations and input/output data structures

(e.g., WSDL and WSCL), and ii) allows other ft_units to coordinate and assemble with it. This component also registers the metadata representing the fault tolerance property of each ft_unit. Service Directory performs a matching of user's preferences and generates the set S of ft_units that satisfy p_c.

- *Composition Engine*: It receives an ordered set of ft_units from the service directory as input, and generates a comprehensive fault tolerance solution ft_sol as output. In terms of service oriented architectures, the composition engine is a web service orchestration engine that exploits BPEL constructs to build a fault tolerance solution.
- *Evaluation Unit*: It monitors the composed fault tolerance solutions at runtime using the validation function and the set of rules defined corresponding to each ft_sol. The interface exposed by web services (e.g., WSDL and WSCL) allows the evaluation unit to validate the rules. If a violation is detected, the evaluation unit updates the present attribute values in the metadata, otherwise, the service continues uninterrupted.

A set of components that provide a complementary support to fault tolerance mechanisms are included in FTM. These components affect the quality of service and support ftSP in satisfying user's requirements and constraints. Figure 5 illustrates the overall architecture of the FTM. A brief discussion on the functionality of each component is as follows:

- *Client Interface*: This component provides a specification language which allows clients to specify and define their requirements.
- *Resource Manager*: This component maintains a consistent view of all computing resources in the Cloud to (i) efficiently perform resource allocation during each user request and (ii) avoid over provisioning during failures. Resource manager monitors the working state of physical and virtual resources, maintains a database of inventory and log information, and a graph representing the topology and working state of all the resources in the Cloud.
- *Replication Manager*: This component supports the replication mechanisms by invoking the replicas and managing their execution as defined in the ft_unit. The set of replicas that are controlled by a single replication mechanism is denoted as a replica group. The task of the replication manager is to make the user perceive a replica group as a single service, and ensure that each replica exhibits correct behavior in the fail-free periods.
- *Fault Detection/Prediction Manager*: This component provides FTM with failure detection support at two different levels. The first level offers failure detection globally, to all the nodes in the Cloud (infrastructure-centric) and the second level provides support only to detect failures among individual replicas in each replica group (user application-centric). This component supports several well-known failure detection algorithms (e.g., gossip-based protocols, heartbeat protocol) that are configured at runtime according to user's preferences. When a failure is detected in a replica, a notification is sent to fault masking manager and recovery manager.

- *Fault Masking Manager*: The goal of this component is to support ft_units that realize fault masking mechanisms so that occurrence of faults in the system can be hidden from users. This component applies masking procedures immediately after a failure is detected so as to prevent faults from resulting into errors.
- *Recovery Manager*: The goal of this component is to achieve system-level resilience by minimizing the downtime of the system during failures. It supports ft_units that realize recovery mechanisms so that an error-prone node can be resumed back to a normal operational mode. The support offered by this component is complementary to that of the failure detection/prediction manager and fault masking manager, when an error is detected in the system. FTM maximizes the lifetime of the Cloud infrastructure by continuously checking for occurrence of faults and by recovering from failures.
- *Messaging monitor*: This component extends through all the components of FTM and offers the communication infrastructure in two different forms: message exchange within a replica group, and inter-component communication within the framework. Messaging monitor integrates WS-RM standard with other application protocols to ensure correct messaging infrastructure even in the presence of failures. This component is therefore critical in providing maximum interoperability, and serves as a key QoS factor.

For example, consider that at the start of the service, the resource manager generates a profile of all computing resources in the Cloud and identifies five processing nodes $\{n_1, \ldots, n_5\} \in N$ with the network topology represented in Figure 6a. Further consider that the FTMKernel, upon gathering the user's requirements from the Client Interface, chooses a passive replication mechanism for the e-commerce service. Based on the chosen fault tolerance mechanism (i.e., the set of ft_units that realize the envisioned passive replication scheme), FTMKernel requires the following conditions to be satisfied: (i) the replica group must contain one primary and two backup nodes (ii) the node on which the primary replica executes must not be shared with any other VM instances, (iii) all the replicas must be located on different nodes at all times, and (iv) node n_5 must not allow any user-level VM instance (rather it should be used only to run system-level services such as monitoring unit). An overview of the activities performed by each supporting component in the FTM is as follows:

- The replication manager (RM) selects the node n_1 for the primary replica and nodes n_3 and n_4 respectively for two backup replicas so that a replica group can be formed (see Figure 6b). Assume that the replication manager synchronizes the state between the replicas by frequently checkpointing the primary and updating the state of backup replicas.
- The messaging manager establishes the infrastructure required for carrying out the checkpointing protocol, and forms the replica group for the e-commerce service (see Figure 6c).

- Assume that the service directory selects a proactive fault tolerance mechanism. As a consequence, the failure detection/prediction manager continuously gathers the state information of nodes n_1, n_3 and n_4, and verifies if all system parameter values satisfy threshold values (e.g., physical memory usage of a node allocated to a VM instance must be less than 70% of its total capacity).
- When the failure detection/prediction manager predicts a failure in node n_1, it invokes the fault masking ft_unit that performs a live migration of the VM instance. The entire OS at node n_1 is moved to another location (node n_2) so that e-commerce customers do not experience any impact of the failure.
- Though the high availability goals are satisfied using the fault masking manager, the IaaS may be affected since the system now consists of four working nodes only. Therefore, FTM applies robust recovery mechanisms at node n_1 to resume it to normal working state, increasing the system's overall lifetime.

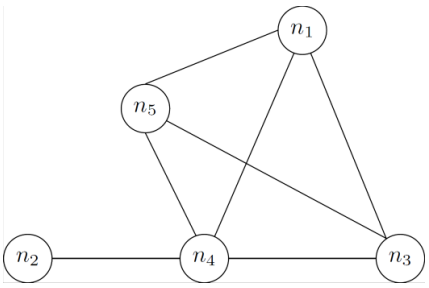

Fig. 6a Resource Graph

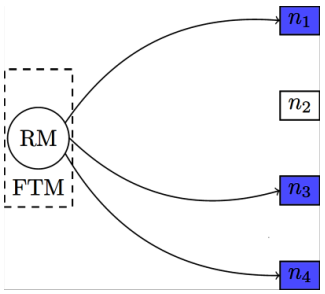

Fig. 6b Nodes selected by Replication Manager

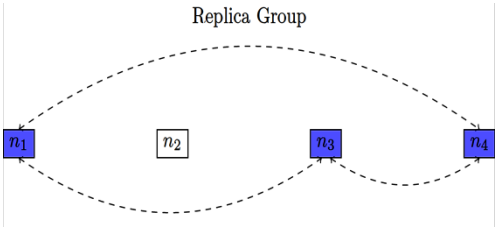

Fig. 6c Messaging Infrastructure created (forms a replica group)

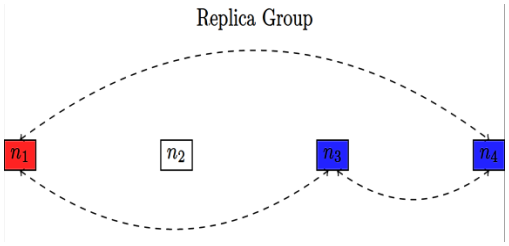

Fig. 6d Failure detected at node n1

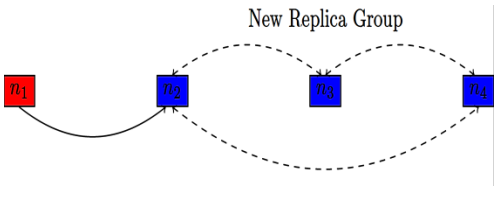

Fig. 6e Fault masking performed – VM instance migrated to node n2

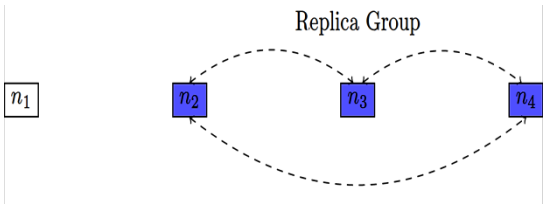

Fig. 6f Recovery Manager brings back node n1 to working state

Using the FTM framework, the notion of providing fault tolerance as a service can therefore be effectively realized for the Cloud computing environment. Based on the delivery scheme that FTM uses, users can achieve high levels of reliability and availability for their applications without having any knowledge about the low-level mechanisms, and dynamically change the fault tolerance properties of its applications (based on the business needs) at runtime.

8. Conclusions

Fault tolerance and resilience in Cloud computing are critical to ensure correct and continuous system operation. We discussed the failure characteristics of typical Cloud-based services and analyzed the impact of each failure type on user's applications. Since failures in the Cloud computing environment arise mainly due to crash faults and byzantine faults, we discussed two fault tolerance solutions, each corresponding to one of these two classes of faults. The choice of the fault tolerance solutions was also driven by the large set of additional properties that they offer (e.g., generality, agility, transparency and reduced resource consumption costs).

We also presented an innovative delivery scheme that leverages existing solutions and their properties to deliver high levels of fault tolerance based on a given set of desired properties. The delivery scheme was supported by a conceptual framework which realized the notion of offering fault tolerance as a service to user's applications. Due to the complex nature of Cloud computing architecture and difficulties in realizing fault tolerance using traditional methods, we advocate fault tolerance as a service to be an effective alternative to address user's reliability and availability concerns.

ACKNOWLEDGEMENTS

This work was supported in part by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4).

REFERENCES

[A.Axis2] Apache Axis2/Java: http://axis.apache.org/axis2/java/core/

[ABL.2008] N. Ayari, D. Barbaron, L. Lefevre, P. Primet, "Fault Tolerance for Highly Available Internet Services: Concepts, Approaches and Issues", IEEE Communications Surveys Tutorials, vol. 10, no. 2, pp. 34—46, 2008

[ADJ.2012] C. Ardagna, E. Damiani, R. Jhawar, V. Piuri, "A model-based approach to reliability certification of services", in Proc. of DEST-CEE'12, Campione d'Italia, Italy, 2012, pp. 1—8

[AFG.2009] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A.

Rabkin, I. Stocia, M. Zaharia, “Above the Clouds: A Berkeley View of Cloud Computing”, EECS Dept., University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009

[AWS.EC2] Amazon Elastic Compute Cloud: http://aws.amazon.com/ec2

[AWS] http://aws.amazon.com/

[Azure] http://www.windowsazure.com/en-us/

[BS.1995] T. C. Bressoud, F. B. Schneider, “Hypervisor-based Fault Tolerance”, in Proc. of SOSP’95, Colorado, USA, 1995, pp. 1—11

[CFH.2005] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, “Live Migration of virtual machines”, in Proc. of NSDI’05, Boston, MA, USA, pp. 273—286

[Cisco.2004] Cisco Load Balancing Data Center Services: https://learningnetwork.cisco.com/servlet/JiveServlet/previewBody/3438-102-1-9467/cdccont_0900aecd800eb95a.pdf

[CL.1999] M. Castro, B. Liskov, “Practical Byzantine Fault Tolerance”, in Proc. of OSDI’99, New Orleans, LA, USA, 1999, pp. 173—186

[CLM.2008] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, “Remus: High Availability via Asynchronous Virtual Machine Replication”, in Proc. of NSDI’08, San Francisco, CA, USA, pp. 161—174

[DFJ.2010] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, P. Samarati, “Encryption-Based Policy Enforcement for Cloud Storage”, in Proc. of ICDCSW’10, Genoa, Italy, 2010, pp. 42—51

[DP.1989] F. Distante, V. Piuri, “Hill-Climbing Heuristics for Optimal Hardware Dimensioning and Software Allocation in Fault Tolerant Distributed Systems”, IEEE Transactions on Reliability, vol. 38, no. 1, pp. 28—39, 2009

[Euc.CM] Eucalyptus Cloud Manager: http://www.eucalyptus.com

[FRM.2012] E. Feller, L. Rilling, C. Morin, “Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds”, in Proc. of CCGrid’12, Ottawa, Canada, 2012, pp. 482—489

[GCP] https://cloud.google.com/

[GJN.2011] P. Gill, N. Jain, N. Nagappan, “Understanding Network Failures in Data Centers: Measurement, Analysis and Implications”, ACM Computer Communication Review, vol. 41, no. 4, pp. 350—361, 2011

[GY.2010] R. Guerraoui, M. Yabandeh, “Independent faults in the Cloud”, in Proc. of LADIS’10, Zurich, Switzerland, 2010, pp. 12—17

[HB.2009] U. Helzle, L. A. Barroso, “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines”, 1st ed. Morgan and Claypool Publishers, 2009

[HDL.2011] F. Hermenier, S. Demassey, X. Lorca, “Bin repacking scheduling in Virtualized datacenters”, in Proc. of CP’12, Perugia, Italy, 2011

[HH.1997] A. Heddaya, A. Helal, “Reliability, Availability, Dependability and Performability: A User-Centered View”, Boston, MA, USA, Tech. Rep., 1997

[HLM.2011] F. Hermenier, J. Lawall, J.M. Menaud, G. Muller, “Dynamic Consolidation of Highly Available Web Applications”, INRIA, Tech. Rep. RR-7545, 2011

[HS.1993] M. Hiltunen, R. Schlichting, “An Approach to Constructing Modular Fault-Tolerant Protocols”, in Proc. of SRDS’93, Princeton, NJ, USA, 1993, pp. 105—114

[JP.2012] R. Jhawar, V. Piuri, “Fault Tolerance Management in IaaS Clouds”, in Proc. of ESTEL’12, Rome, Italy, 2012 (to appear)

[JPS.2012a] R. Jhawar, V. Piuri, M. Santambrogio, “A Comprehensive Conceptual System-Level Approach to Fault Tolerance in Cloud Computing”, in Proc. of IEEE SysCon’12, Vancouver, BA, Canada, 2012, pp. 1—5

[JPS.2012b] R. Jhawar, V. Piuri, M. Santambrogio, “Fault Tolerance Management in Cloud Computing: A System-Level Perspective”, IEEE Systems Journal, 2012 (to appear)

[KAD.2009] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance", in ACM Transactions on Computer Systems, vol. 27, no. 4, pp. 7.1—7.39, 2009

[KMT.2009] S. Kim, F. Machinda, K. Trivedi, "Availability Modeling and Analysis of Virtualized System", in Proc. of PRDC'09, Shanghai, China, 2009, pp. 365—371

[KYW.2010] G. Koslovski, W. L. Yeow, C. Westphal, T.T. Huu, J. Montagnat, P. Vicat-Blanc, "Reliability Support in Virtual Infrastructures", in Proc. of CloudComm'10, Indianapolis, IN, USA, 2010, pp. 49—58

[MFD.2011] K. Mills, J. Filliben, C. Dabrowski, "Comparing VM-Placement Algorithms for on-demand Clouds", in Proc. of CLOUD'11, Washington, USA, 2011, pp. 91—98

[MKM.2010] F. Machinda, M. Kawato, Y. Maeno, "Redundant Virtual Machine Placement for Fault Tolerant Consolidated Server Clusters", in Proc. of IEEE/IFIP NOMS'10, Osaka, Japan, 2010, pp. 32—39

[MLV.2011] Y. Mao, C. Liu, J. E. van der Merwe, M. Fernandez, "Cloud resource orchestration: A data-centric approach", in Proc. of CIDR'11, Asilomar, CA, USA, 2011, pp. 241—248

[P.1994] V. Piuri, "Design of Fault-tolerant distributed Control Systems", IEEE Transactions on Instrumentation and Measurement, vol. 43, no. 2, pp. 257—264, 1994

[S.2004] B. Selic, Fault tolerance techniques for distributed systems: http://www.ibm.com/developerworks/rational/library/114.html

[SD.2010] P. Samarati, S. De Capitani di Vimercati, "Data Protection in Outsourcing Scenarios: Issues and Directions", in Proc. of ASIACCS'10, Beijing, China, 2010, pp. 1—14

[STT.2008] W.E. Smith, K.S. Trivedi, L.A. Tomek, J. Ackaret, "Availability Analysis of Blade Server Systems", IBM Systems Journal, vol. 47, no. 4, pp. 621—640, 2008

[TBH.2012] A. Tchana, L. Broto, D. Hagimont, "Approaches to Cloud computing fault tolerance", in Proc. of CITS'12, Amman, Jordan, 2012, pp. 1—6

[TSK.2008] Y. Tamura, K. Sato, S. Kihara, S. Moriai, "Kemari: Virtual Machine Synchronization for Fault Tolerance", in Proc. of USENIX Annual Technical Conference, Boston, MA, USA, 2008

[UCH.2011] A. Undheim, A. Chilwan, P. Heegaard, "Differentiated Availability in Cloud Computing SLAs", in Proc. of Grid'11, Lyon, France, 2011, pp. 129-136

[VN.2010] K. Vishwanath, N. Nagappan, "Characterizing Cloud Computing Hardware Reliability", in Proc. of SoCC'10, Indianapolis, IN, USA, 2010, pp. 193—204

[WS.BPEL] OASIS Web Services Business Process Execution Language Version 2.0 (WS-BPEL): http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[WS.RM] OASIS Web Services Reliable Messaging (WS-RM): https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm

[WSV.2011] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, E. Cecchet, "ZZ and the art of Practical BFT Execution", in Proc. of EuroSys'11, Salzburg, Austria, 2011, pp. 123—138

[YMV.2003] J. Yin, J.P. Martin, A. Venkataramani, L. Alvisi, M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services", in Proc. of SOSP'03, New York, NY, USA, 2003, pp. 253—267

[ZMM.2010] W. Zhao, P. Melliar-Smith, L. Moser, "Fault Tolerance Middleware for Cloud Computing", in Proc. of CLOUD'10, Miami, FL, USA, pp. 67—74